第1章

# LSI開発における検証の重要性を考える

## ―不具合をなくし開発期間を短縮するために

古川 寛

**近年では，LSI開発期間の大部分が検証に費やされるようになっている．検証は，LSIが正しく動作することを保証するために必須であり，ASICにかぎらずFPGAをターゲットにする開発であってもとても重要だからである．ここでは，LSI開発における検証の重要性を考える．特に機能検証について注目する．LSI開発における手戻り要因の多くが所定の機能的な不具合にあるという．（編集部）**

機能検証が大きくクローズアップされてきています．

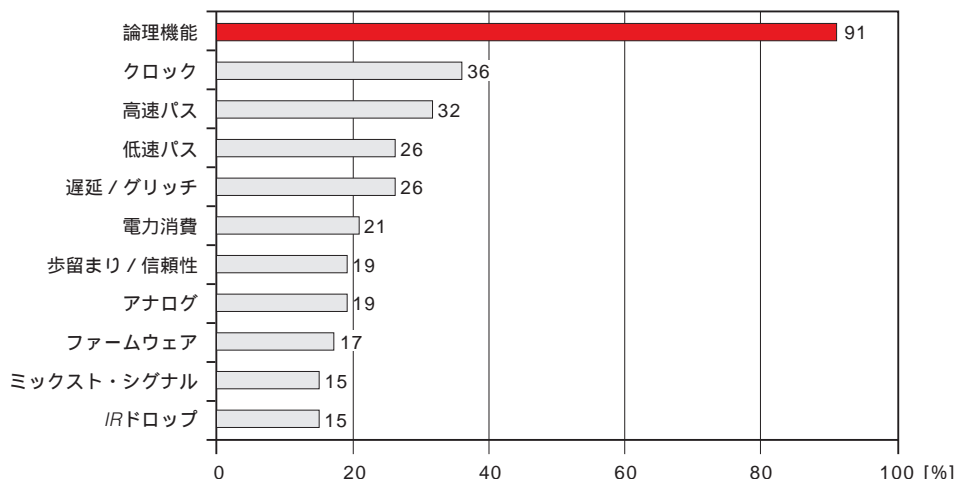
LSIの大規模化に伴い，一つのLSIの中により多くの機能を実装するようになったことが直接の要因です．検証のための工数は，機能の数に比例して増えます．また，関連する機能や信号などの組み合せが増えると，指数関数的に増大することになります．これがフロントエンド設計で最大の問題になっています．

LSIの開発コストの増大も理由の一つです．LSIの製造プロセスが90nm世代になり，マスク代は数千万円にも跳ね上がりました．このため，リスピン（不具合のためのマスク再設計）の費用が半端でなくなっています．リスピンの原因の91%は，論理機能的な不具合だったという調査結果もあります（**図1**）．2004年は70%程度でした．リスピンをなくすためには，検証に注力するしかありません．

これからのLSI開発で考えなければいけないのは，機能的な不具合の撲滅と，検証期間の改善の二つです．具体的には，何を検証すべきか（検証すべき項目のリストアップ）と，どう検証すべきか（検証手法の明確化）ということです．

本稿では，機能検証のあり方と今後について考えます．Verilog HDLによるRTL（register transfer level）設計を

**図1 リスピンの原因**

LSIの最初の試作時に発生した不具合の原因をまとめたものである．原因のうち91%は，論理機能的な不具合だった．この割合は，2004年には70%程度であり，増加している．出展：Collett International, January 2005

**KeyWord** 機能検証，リスピン，論理シミュレータ，テストベンチ，検証仕様書，検証プラン，カバレッジ，第三者検証，アサーション

前提としています．

## 1．LSI開発フローにおける機能検証

今日のLSI開発フローを図2に示します．基本的には，上から下へと進みます．もちろん，実際には問題が見つかれば前工程への手戻りが発生します．フロー通りに開発を進められるかどうかについて，あらかじめ試行しておくことも必要です．

### ● 論理的機能の作り込みにおける山場は機能検証

フローの前半は論理的な設計で機能を作り込みます．まず，仕様を決定し，その仕様に基づいて機能をRTLで設計します．次に，設計した機能が仕様通りに正しく動作するかを検証します．一番の山場が機能検証になります．LSIが正しく動作するかどうかは，この機能検証にかかっていると言っても過言ではありません．機能検証を通ったら，論理合成ツールでRTLをゲート・レベルに落とし込み，これにテスト回路を自動挿入してネットリストを作り上げます．

フロー後半は，物理的な設計でタイミングを作り込みます．自動レイアウト・ツールを使い，論理セルの配置とセル間の配線を行います．また，物理情報を元にタイミングの検証を行います．

LSIが完全に動作するためには，機能とタイミングを満足させなければなりません．設計（作り込み）ももちろん大事ですが，きちんと検証することが不具合をなくすためには重要です．

### ● 開発期間の多くが機能検証で費やされる

機能を作り込むためにRTL設計した後，設計した機能が仕様通りに正しく動作するか確認をすることを「機能検証」と言います（図3）．

実際の開発では，この工程でイタレーション（繰り返し）が多く発生します．また，機能検証の最後には，最終版のRTLですべての機能をフルチェックすることになります．ある一つの機能の不具合を修正したときに，その機能についてのみ改めて検証すればよいと思う方もいらっしゃるかもしれません．こうした考えは，検証途中においては間違いではありません．しかし，その機能の修正がほかの機能に悪影響を及ぼす（不具合をひき起こす）こともあります．

このため，LSIの集積度が上がり，多くの機能が実装されるようになった今，RTL担当者の仕事の大半は機能検証になっています．意外かもしれませんが，RTL設計よりはるかに時間を要するようになっているのです．機能検証は，どこまでやればよいのかが主観的でもあります．客観的な判断が困難なため，人次第で必要時間が倍になったり半分になったりします．

### ● 機能検証では論理シミュレータを使う

機能検証はソフトウェアのシミュレータを使うのが一般的です（図4）．シミュレータはLSIのふるまいをソフトウェア上で模擬するツールです．

シミュレータを使うに当たり，設計したRTLコードに加えて「テストベンチ」が必要になります（図5）．テストベンチとは，RTLコードをDUT（device under test；検証対象物）と見立てたとき，入力信号のパターンを与えて出力信号のパターンを観測するための記述です．RTLコードを

**図2 LSI開発フロー**
上から下へと流れるが，問題が見つかれば前工程への手戻りが発生する．

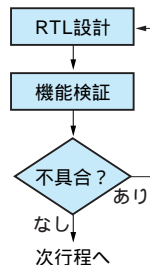



**図3 機能検証のフロー**
機能を作り込むためにRTL設計した後，設計した機能が仕様どおりに正しく動作するか確認することを「機能検証」と言う．不具合がなくなるまで繰り返される．

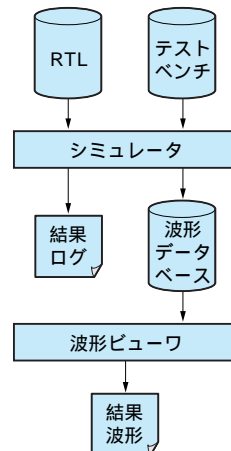



**図4 機能検証の環境**
ソフトウェアのシミュレータを使うのが一般的．

Verilog HDLで書く場合は，テストベンチもVerilog HDLで書くのが一般的です．

シミュレータには，RTLコードとテストベンチを入力します．シミュレータは，テストベンチに書かれた指示に従ってRTLコードで記述されたLSIの動作を模擬し，結果を出力します（図6）．出力結果を確認することにより，設計したRTLコードが仕様通りに正しく動作しているかを判断します．

シミュレーション結果の確認は，ログの目視や期待値比較などによって行います．また，テキストでは確認しにくいような場合は，GUI（graphical user interface）を使った波形表示で目視確認することもあります（図7）．出力信号だけでなく，入力信号や内部ノードも観測することが可能です．ただし，観測点を増やすとシミュレーション速度の低下を招くことにも注意が必要です．

表1に，LSI設計でよく使われているシミュレータの例を示します．LSI設計言語はVerilog HDLだけではないので，サポート言語の異なるシミュレータをリリースしているベンダもあります．また，波形ビューワとしては米国Novas社のDebussyがあまりにも有名ですが，ほとんどのシミュレータに標準で付属する機能の一つになっています．また，単なる波形ビューワだけでなく，HDLコード・ビューワ，回路図ビューワがあり，かつそれらが連携していることで，パワフルなデバッグ環境が実現されます．

機能検証では，ソフトウェアのシミュレータを使うことが最も一般的な方法になっています．外部機器と接続して検証を行わなければならないような場合，実速度が求められるので，FPGAを使ったプロトタイピングなどもよく行われています．

## 2. 機能検証のポイント

ここでは機能検証の肝である「何をどう検証すべきか」について説明します．

機能検証の詳細フローを図8に示します．効率的で確実な機能検証のためには，検証仕様の作成と検証プランの決定がポイントになります．

### ● 検証仕様書の作成 － 何を検証するのか考える

基本的には機能仕様書から検証仕様書を作成します．これが機能検証の“神様”であり起点なので，最も重要と言えます．

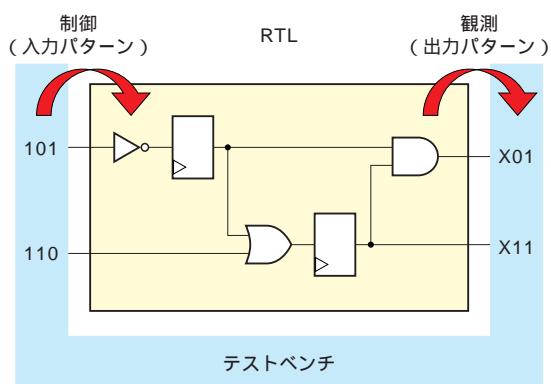



**図5　テストベンチの役割**

RTLコードをDUT（device under test；検証対象物）と見立てたとき，入力信号のパターンを与え，出力信号のパターンを観測するための記述がテストベンチである．

```
reset=0  count=11
reset=1  count=00
reset=0  count=01
reset=0  count=10
reset=1  count=00
reset=0  count=01
reset=0  count=10
```

**図6**
**シミュレーション結果の例**

シミュレータは結果をテキスト形式で出力することが多い．

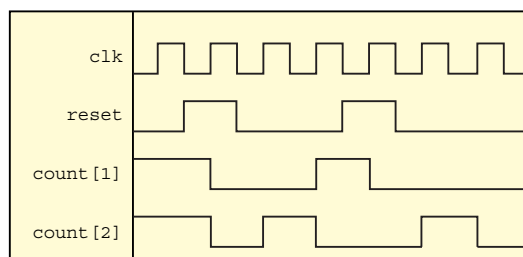



**図7　シミュレーション結果の波形表示**

GUI（graphical user interface）を持つシミュレータでは波形表示もできる．また，波形表示を行うための専用ツールもある．

**表1　シミュレータの例**

| EDAベンダ名 | プラットホーム名 | ツール名 |
|---|---|---|
| 米国Synopsys社 | Discovery | VCS |
| 米国Cadence Design Systems社 | | NC-Verilog |
| | Incisive | IUS |
| 米国Mentor Graphics社 | | Model Sim |
| | Questa | Questa System Verilog |
| | | Questa AFV |

まず，機能について検証すべき項目を抽出します(**表2**)．米国Microsoft社の表計算ソフトウェアExcelなどを使って表形式で作成するのが一般的です．

検証漏れをなくすために重要なポイントは，

- 検証すべき項目がすべて網羅されているか
- 検証すべき条件は明確か

です．検証すべき項目が記載されていなければ，検証されないことになります．検証すべき条件があいまいだと，本来の機能を検証できていなかったということになるかもしれません．

検証すべき条件については，

- コーナ・ケース(境界条件や組み合せ条件など)をいかに想定できるか
- 動くべきでないときに動いていないことの確認や異常時の動作まで想定できるか
- 実チップで不具合が発生したとき，検証時の状況が正確に再現可能か

についても考えておく必要があります．

検証仕様書を作ることで，検証工数のボリュームを見積ることもでき，また進捗管理にも使えるようになります．

### ● 検証プランの作成 —— どう検証するのか考える

検証プランとしては以下の2点が肝になります．

- どの階層においてどの検証を行うのか
- どう結果確認をするのか

#### 1) どの階層においてどの検証を行うか

今日のSOC(system on a chip)は多くのモジュール(module)で構成されています．機能検証にあたっては，どの階層においてどの検証を行うかが重要になります．なぜなら，テストベンチは想定した階層でしか使えないからです(**図9**)．

トップ階層だけで検証できれば，それに越したことはありません．しかし，大規模な設計おいてトップ階層から検証しようとしても，シミュレーション速度の低下を招くことになります．また，制御や観測が困難になります．制御性を上げるために内部ノードに初期値(initial)や固定値(force/release)を与えると，実動作と異なってしまいます．これでは間違った検証を行ってしまうことがあります．特別な理由がない限りトップ階層だけで行うことは推奨できません．

逆にすべてモジュール単位で検証するためには，全モジュール分のテストベンチが必要になります．検証対象のモジュール数やモジュールの規模にもよりますが，SOCのような大規模LSIであれば膨大な数になります．モジュールの規模が小さすぎる場合はまともな機能検証が難しくなります．

現実的には，下位である程度の機能ブロックごとにテストベンチを作って，その機能単体はそこでしっかり検証し，さらに上位でそれらの機能ブロックをまとめたテストベンチを作って，各ブロックのつなぎや機能の組み合わせだけを検証して補完することになります．

これをきちんと管理しておかないと，検証漏れや2重検

**表2 検証すべき項目の抽出例**

| No. | 大項目 | 中項目 | チェック項目 | 判　定 |
|---|---|---|---|---|
| 1 | $I^2C$ | read | … | ○ |
| 2 | | write | … | |
| 3 | … | … | … | |

**図8 機能検証の詳細フロー**
効率的で確実な機能検証のためには，検証仕様の作成と検証プランの決定がポイントになる．

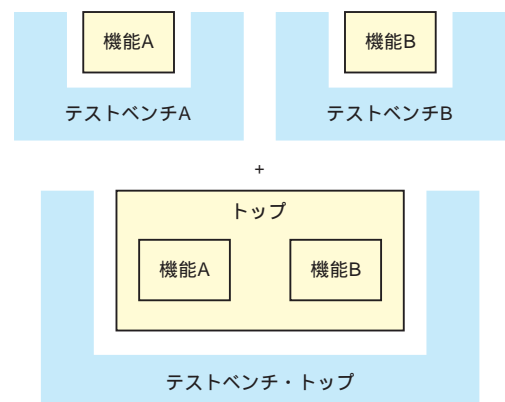
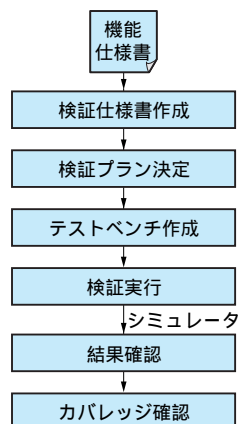



**図9 どの階層においてどの検証を行うか**
どの階層においてどの検証を行うかが重要．2レベルだけの検証例を示しているが，数レベルになる場合もある．

証による工数の無駄遣いが発生することになりかねません．なお，図9では2レベルだけの検証例を示していますが，数レベルになる場合もあります．

**2）どう結果確認をするのか**

検証結果の確認は，ログの目視や期待値比較などによって行います．

目視では，

- 0/1などによる数値の確認
- 自動チェック埋め込みによるエラー・メッセージなどの確認
- 波形エディタによる波形の確認

などを行います．ここも主観的なあいまいさの入り込む部分であり，特に波形エディタではどこの波形を見るのかまで明記すべきです．

### ● 機能検証における客観的判断材料はカバレッジ

図9の結果確認で問題なければ，機能検証は終了とも言えます．しかし，ここまでで客観的な判断材料はなにもありませんでした．機能検証において唯一とも言える客観的な指標が，「カバレッジ（検証網羅性）」です．

カバレッジには大きく2種類があります．いずれも，基本的には100%でなければなりません．

- コード・カバレッジ：RTL記述上どれだけ動作したか（母数は自動的にツールが定義する）
- 機能カバレッジ：機能がどれだけ検証されたか（母数はユーザが定義する）

**1）コード・カバレッジ**

コード・カバレッジは，通常はシミュレータに付加されている機能の一つです．シミュレータの設定をONにすることで採取できます．ただし，ONにするとシミュレータの速度低下を招くことに注意が必要です．

コード・カバレッジでは，以下の4種類の網羅性を確認します．

**表3　コード・カバレッジと機能カバレッジ**

| コード・カバレッジ | 機能カバレッジ | 検証状況 | バグが潜んでいる可能性 |
|---|---|---|---|
| 低い | 低い | 検証が不十分 | 大 |
| 低い | 高い | 検証項目の洗い出しが不十分 | 中 |
| 高い | 低い | コーナ・ケースや複雑な組み合わせが不十分 | 小 |
| 高い | 高い | 検証が十分 | なし？ |

- ライン・カバレッジ（ブロック・カバレッジ）：このライン（代入文）は実行されましたか？
- コンディション・カバレッジ（エクスプレッション・カバレッジ）：if文の条件文はすべての組み合せをとりえましたか？
- トグル・カバレッジ：`wire`，`reg`宣言された信号はトグル（変化）しましたか？
- FSMカバレッジ：全ステートを取りえましたか？起こりえるステート遷移はしましたか？

コード・カバレッジは比較的低レベルなので，検証網羅性の指標としては陳腐であり，あくまで必要最低限と言えるものです．にもかかわらず，もしコード・カバレッジを100%にできないとしたら，冗長な記述が残っていることが多いようです．疑似エラーに埋もれて真の未検出個所を見逃さないためにも，冗長な記述は削除すべきです．

**2）機能カバレッジ**

機能カバレッジは，検証仕様書に結果が入ったものをどれだけ確認したかの指標といえます．検証項目数を分母に，検証済で結果OKの項目数が分子になります．これこそ，検証網羅性の指標として一番適当であると言えるでしょう．しかし，検証項目をユーザが定義していることから，母数に主観が入り込み，あいまいです．

そのあいまいさを補完する意味でも，コード・カバレッジと機能カバレッジをともに使い，真の検証網羅性をあげていくことが重要です（表3）．また，一度で100%になることは少ないので，その場合は未検証個所を洗い出し，テスト・パターンを追加することで，100%になるまで繰り返すことになります．

## 3．検証品質を高める

### ● 第三者検証

従来の開発では，設計者が検証を行う場合がほとんどでした．設計した人が検証もやるというのは，工数的には効率が良くなります．一つの機能について，設計者と検証者が異なると，その機能を両者が理解しなければならないためです．2度手間とも言えます．

ところが最近になって，この2度手間をあえてやろうという動きがあります．これは，「第三者検証」と呼ばれています．すなわち，設計者と検証者を分けるのです．

主観だらけの世界の中で，思い込みによる不具合をなく

すためには，設計には携わっていない第三者が異なった視点で検証することが必要です．なぜなら，設計者が思い込みで作った仕様を元に検証するのであれば，その思い込みが間違いだと気付くところがないからです．

第三者検証を行えば，品質は確実に向上するはずです．しかし，工数も確実に増大してしまいます．検証者もアプリケーション（機能）を理解しなければならないからです．機能仕様書としてテストベンチが作れるレベルのタイミング・チャートなど詳細が必要になります．

いちばん工数を要するのは“神様”である検証仕様書です．設計者が検証もやる場合は，検証仕様書もその人が書くわけなので，あいまいな記述であっても本人の頭の中にはやることがそれなりに明確になっています．ところが，そのあいまいな記述は第三者には理解できません．これを，第三者が見ても理解できるような，一意の記述にしなければなりません．

しかし，ここが第三者検証の一番のメリットでもあるのです．一意の記述にするということは，あいまいさが除かれることを意味するからです．

また，検証仕様書さえしっかりできてしまえば，検証を他人だけでなく他社にも委託できます．責任分担も明確だからです．委託された側も，基本的には人とシミュレータだけ用意すれば可能です．

昨今，機能検証専門会社や機能検証請負サービスが出てきました．従来の機能検証不足に加えて第三者検証の必要性の高まりによって，はやってきています．おまけに，高い値段で受注できるそうです．派遣でも，ほかの業務より1.5～2倍は高いと言います．

ところで，検証者がバグを発見したらどうすればよいのでしょうか．当たり前ですが，設計者に戻すことになります．検証者はバグを改修することはできないからです．検証者はRTL記述をまったく見ない，ブラックボックス検証しか行いません．あくまで入力信号と出力信号のみを使って検証を行います．バグはRTL内部に潜んでいるので，この改修を行うためには，RTL内部を見るホワイトボックス検証を，RTL内部を唯一理解している設計者が行うしかないのです．

ちなみに，設計者がまったく検証せずに，検証者にRTLを渡すということはありません．機能を作り込んでいく中でのホワイトボックス検証で，ある程度のデバッグを行った後に，クリーンになったものを検証者に渡します．バグだらけの状態で渡しても設計者と検証者のイタレーションが頻繁になり，真のメリットに辿りつかないうちに開発を終えてしまうことになりかねないからです．

### ● 検証とテストベンチのための言語

これまで筆者は，設計も検証もすべてVerilog HDLを使っていました．最近になって，これらがそれぞれ分離してきています．図10に示す言語はすべて，すでにIEEE（Institute of Electrical and Electronics Engineers,Inc.）における標準化が終了しています．

分野特化で専用になれば，その分野ではより強力になるのが普通です．これまでもVerilog HDLで記述できたことであっても，専用の言語であれば少ない記述で平易に所望の内容が書けるようになります．これらの新しい言語に，シミュレータも対応してきています．

検証（アサーション）言語は，主に観測（出力）側に用います（図11）．期待する動作を記述し，常に監視し，違反したらログにエラーを出力します．例えば，これまで期待する動作を波形で見ていたのに比べて，目視というあいまいさをなくし，かつ見落としなしにチェックできることがメリットになります．検証結果の確認が明確になるだけでなく，効率も上がります（目視不要になるため）．さらには，RTLバージョン変更後の繰り返し検証や，次製品への再利用の際にも有用です．

テストベンチ言語は主に制御（入力）側に用います．例えば，制約付きのランダム・パターンを生成し，入力ピンに印加する記述を行います．純粋なランダム・パターンだと

昔

| 設計言語 | Verilog HDL（Verilog 1995（IEEE 1364-1995）, Verilog 2001（IEEE 1364-2001）） |
|---|---|
| 検証（アサーション）言語 | |
| テストベンチ言語 | |

今

| 設計言語 | SystemVerilog | (IEEE 1800) | |
|---|---|---|---|
| 検証（アサーション）言語 | SystemVerilogAssertion（SVA） | (IEEE 1800) | PSL（IEEE 1850） |
| テストベンチ言語 | SystemVerilogTestBench（SVTB） | (IEEE 1800) | e（IEEE 1647） |

**図10 言語の分野特化**

従来Verilog HDLによる開発では，設計，検証，テストベンチのすべてでVerilog HDLを使用していた．しかし最近になって，分野特化の言語が登場している．

**図12**
**さまざまな機能検証手法**
**機能検証ではシミュレーションが広く用いられているが，実行速度，網羅性によりさまざまな手法がある．実際には検証環境の立ち上げの手間や困難さのほかコストも異なるので，適切な選択を行う必要がある．**

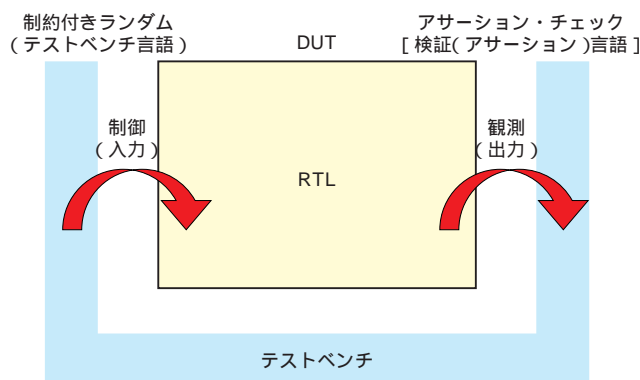
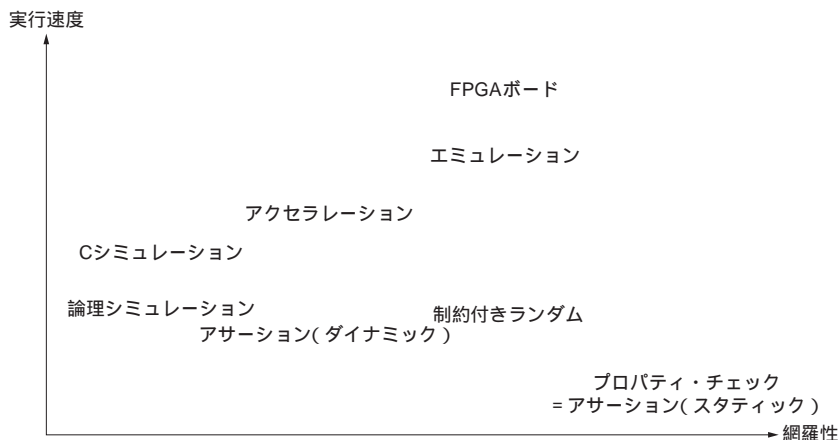



**図11　検証（アサーション）言語とテストベンチ言語**
**検証（アサーション）言語は，主に観測（出力）側に用いる．テストベンチ言語は，主に制御（入力）側に用いる．二つを併用することでより効果が上がると考えられる．**

無駄に時間を消費してしまうので，制約付きという点がみそです．しかしながらランダムという性質上，コーナ・ケースや組み合せを検証することが期待できます．

これらの言語を使った手法は，単品でも効果がありますが，併用することでより効果が上がると考えられます．検証においては制御と観測が重要であり，制御性を上げるテストベンチ言語と観測性を上げる検証（アサーション）言語の両方を使うことで大きなメリットを生み出すからです．

## ● シミュレーション以外の機能検証手法

機能検証はシミュレータだけで行うことではありません．**図12**のような手法もあり，それぞれメリットとデメリットがあります．キーワードは，実行速度，網羅性に加え，立ち上げの手間や困難さです．これらはアプリケーション（機能）と言うよりは，どういう検証をやりたいかという目的によって，間違いのない選択を行うことで効果を発揮できます．

特にポイントになるのは以下の2点です．

- 制御系なのか，データ・パス系なのか
- 実動作速度が必要か否か（外部機器との接続は必要か否か）

## ● まとめ

RTL設計ともども機能検証も，基本的には楽しい仕事だと筆者は思います（時間に追いまくられなければ）．LSI設計エンジニアとして，作ったモノが動く様子を確認するのは，モノ作りとしてプリミティブな喜びを感じます．

また，どんなに設計自動化されても，一番最後まで“人”的な部分が残るのも機能検証だと考えます．これはソフトウェア開発と同じかもしれません．機能カバレッジをみてもよく分かるように，あくまで検証項目をあげるのは人間なのです．

人を変えるのが一番手っ取り早い改善かもしれません．しかし，主観的な要素があまりに強すぎる領域で，野球選手のように「打てるヤツ引っ張ってこい」というようなことをしてしまえば，一般的な組織は破綻してしまうでしょう．何らかの客観的な指標をできる限り持ち込むことで，解決できないものだろうかと，筆者は常々考えています．

中国やインドの設計が勢いづいています．人の数とコストでは，もはや日本は勝てそうにありません．日本が勝てる点は「made in Japan」の品質ではないでしょうか．その意味で，機能検証が差異化のポイントになることを期待してやみません．

**ふるかわ・ひろし**
**NECマイクロシステム（株）**
**hiroshi.furukawa@nms.necel.com**